Random Numbers

Andreas Zeller/Stephan Neuhaus
Lehrstuhl für Softwaretechnik
Universität des Saarlandes, Saarbrücken

SS 2004/CVS Revision: 1.11

The Menu

- What is a Random Number?
- Cracking Random Number Generators
- Entropy/Workload estimation
- *rand*(3) and Netscape's Old Generator

Introduction

This is the first part of a two-part lecture on randomness in security.

This part: Basics, simple generators, breaking generators.

Next part: A sampling of generators.

What is a Random Number?

Is 2 a random number?

We talk about a *sequence* of numbers as being random

Okay, a sequence of random numbers begins $132, 521, 254$. Is 132 a random number?

A sequence of random numbers begins $132, 132, 132$. Is 132 a random number?

Briefly, a sequence is random if you can't predict the $(n+1)$-st element of the sequence even if you know the first n elements of the sequence.

Applications for Random Numbers

- Session keys for encrypted data exchange
- One-time passwords
- Random passwords
- HTTP cookies
- Cryptographic tokens and nonces
- TCP initial sequence numbers

Predictiable “Random” Numbers

- Session key compromised $\Longrightarrow$ conversation readable
- One-time password guessable $\Longrightarrow$ bank account plundered
- HTTP cookie guessable $\Longrightarrow$ identity stolen
- Cryptographic token guessed $\Longrightarrow$ protocol broken
- TCP ISN guessed $\Longrightarrow$ connection hijacked

The C Library’s *rand*(3)

```c
#include <stdio.h> /* For printf(3) */
#include <stdlib.h> /* For srand(3) and rand(3) */

int main(int argc, const char* argv[]) {
  unsigned int seed = atoi(argv[1]);
  int i;

  srand(seed);
  for (i = 0; i < 10; i++)
    printf ("%u\n", rand());
  return 0;
}
```

Looks Random, Doesn’t It?

Results of calling the program with 1074781755, 1074781757, and 1074781758:

Run 1	Run 2	Run 3
706062696	1167387154	1949615244
388317246	16703281	1986396061
1795625833	2027155102	538593506
1641240349	1937794218	492684994
1013505830	731325747	600732415
1048427458	929979607	1414558367
1562911947	2034902343	680342290
836469238	1030204849	1677570512
1567615431	740744080	1408795345
84389751	443114406	1719419442

Predicting *rand*(3)

If we know the seed, we can predict the entire sequence.

If we know that the seeds is actually the number of seconds since Jan 1, 1970 at the time the program was run, the sequence becomes guessable:

We guess that the sequence was generated some time after November 2003.

A non-leap year has about $365 \cdot 24 \cdot 3600 = 31536000$ seconds. (We're not counting leap seconds.)

From 1970 to 2003, there were eight leap years with $366 \cdot 24 \cdot 3600 = 31622400$ seconds.

From January 1, 2003 to October 2003 are $365 - 31 - 30 = 304$ days, or 26265600 seconds.

Breaking *rand*(3) With TOD Seed (1)

We start seeding the generator with November 1, 2003, which is approximately
$(2003 - 1970 - 8) \cdot 31622400 + 8 \cdot 31622400 + 26265600 = 1069804800$
and generate the first few numbers.

If they match the sequence that we already know, we stop. Otherwise, we increment the seed and start again.

How long will that take?

On a 2.6 GHz P4, it takes about 30 seconds

Breaking *rand*(3) With TOD Seed (2)

```
#include <stdio.h>
#include <stdlib.h>

static const unsigned int sequence[] = {
```

```
  706062696, 388317246, 1795625833, 1641240349, 1013505830,
 1048427458, 1562911947, 836469238, 1567615431, 84389751,
};
int main() {
  unsigned int seed = 1069804800;
  int success = 0;
  int tries = -1;

  while (!success && tries < 10000000) {
    int i;

    tries++;
    srand(seed + tries);
    for (i = 0; i < sizeof(sequence)/sizeof(sequence[0]); i++) {
      if (rand() != sequence[i])
        break;
    }
    success = i == sizeof(sequence)/sizeof(sequence[0]);
  }
  if (success)
    printf("Seed is %u\n", seed + tries);
  else
    printf("No success\n");

  return 0;
}
```

The Reason: Not Enough Entropy

“Entropy” is often used by cryptographers.

Generally means the amount of “true randomness” in a sequence.

A sequence of n-bit numbers that consists of only one number has no entropy.

A sequence of n-bit numbers that contains all n-bit numbers in a non-periodic, unforseeable sequence has maximum entropy (n bits per element).

Netscape’s Old SSL Session Key Generator

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

typedef unsigned char uint128[16]; /* A 128-bit value */

uint128 make_key() {
  struct timeval now;
  unsigned long a, b;
  uint128 seed, nonce, key;

  gettimeofday (&now, NULL); /* Local time since Jan 1, 1970 */
```

```
  a = mixbits (now.tv_usec);
  b = mixbits (getpid() + now.tv_sec + (getppid() << 12));
  seed = MD5 (a, b); /* C++ Warning! Compute MD5 of concat of a and b */
  nonce = MD5 (seed++); /* C++ warning! */
  key = MD5 (seed++); /* C++ warning! */

  return key;
}
```

How to Crack Old Netscape SSL Data

A snooper records an SSL-encrypted data exchange between a Netscape browser and a Web server and immediately sets about breaking it.

The general strategy is to try to decrypt the traffic with a guessed key (brute-force).

If the correct key was guessed, the output will be a stream of 7-bit ASCII characters containing lots of known plaintext.

This strategy is *independent of the quality of the cipher*.

If we know the seed, we know the key and the nonce.

How can we guess the seed?

Entropy Estimation For A Snooper

`struct timeval` contains seconds and microseconds

Seconds since Jan 1, 1970 trivial to estimate for snooper (looks at watch; sometimes, exchanged cleartext messages have local time of day!). Estimate skew of at most half a minute: at most 5 bits unknown.

“Microseconds” usually aren’t: clock resolution at millisecond level: 10 bits max.

Process ID and parent process ID are 16-bit numbers. Assume maximum entropy: 32 bits.

Total entropy (= size of search space): 47 bits; average number of keys to try if all are equally likely: 2^{46}. (This number is also known as the *cryptographic work factor*.)

Average Crack Time

Average time to find key *in theory* (at 100,000,000 tests per second) $2^{46}/10^8 \approx 2^{46}/2^{26} = 2^{20}$ seconds, or about two weeks.

Average time to find key *in practice* in 1995 about one minute!

The keys are not equally distributed in the search space: start with the more probable values first, hit the key earlier.

- start with the current time and work backwards;
- start with process ID in the low 1000s;
- and keep the parent process ID less than the process ID.

Ballpark Estimates

These are just back-of-the-envelope calculations that are merely used to give a “ballpark figure”, a rough estimate. We also say “The amount of work needed to find the key by brute-force is on the order of a few weeks”.

Obviously, “a few weeks” is better than “a few seconds”, but in “a few years”, “a few weeks” will just *be* “a few seconds”!

So, to be safe, you should aim for “a few times the age of the universe”, or “a few times the number of electrons in the universe”.

X Windows Security—Not!

An X server can authenticate an X client if the client and the server share a secret.

The most common method is called MIT-MAGIC-COOKIE-1 and works by the server generating a random number that the client presents to the server for authentication.

If the wrong cookie is presented, the authentication fails and the client cannot connect to the server.

No matter that communication between client and server is not encrypted, if I can guess the cookie, I can connect to the server, despite “security”.

Here is some actual old code that generates the secret key:

```
key = rand() % 256;
```

Kerberos V4

```
#include <stdlib.h> /* For random(3) and srandom(3) */
#include <unistd.h> /* For gethostid(2) */

static long counter = 1;

long session_key() {
  srandom (time.tv_usec ^ time.tv_sec ^ getpid()
           ^ gethostid() ^ counter++);
```

```
    return random();
}
```

Netscape-Style Entropy

That's even worse than Netscape, because there is no mixing function involved! (The high-order bits will change much less quickly than the low-order bits.)

You can estimate fairly well the value of `counter` from the number of times the routine is called and the machine's uptime. Let's say 10 unknown bits.

The `gethostid()` is tied to the machine's IP address, plus some other easily guessed stuff: perhaps 10 more bits.

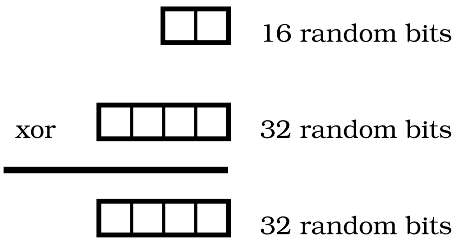
The `tv_sec` member has 5 unknown bits, the `tv_usec` 10 more, `getpid()` has 16.

Total entropy with a proper mixing function would be about 51 bits, which is small enough.

No Mixing Function

The lower 16 bits from `counter` overlap with `getpid()`, so they don't add to the entropy. The upper 16 bits are easily guessed, giving perhaps 10 bits.

If the lower 16 bits are occupied by `getpid()`, only the upper 16 bits of the other variables are relevant.



It's Worse Than Netscape!

The `tv_usec` value will increment in amounts of 1000 (using millisecond resolution), or 2^{10}. That means that only values larger than 2^6 milliseconds will be above the 16 bits, reducing the 10 bits of entropy to 4.

The `tv_sec` value is essentially known; `counter` gives 10 bits, `gethostid()` is easily guessed, giving perhaps also 10 bits.

Total entropy: 24 bits.

It's *Still* Worse Than Netscape!

But the worst thing is... even with a proper mixing function...

The return value from `session_key()` is only 32 bits!

What To Do: Proper Mixing

```
#include <openssl/md5.h>

static void
digest_mix(unsigned char *digest, int random1, int random2) {
  MD5_CTX md5;
  unsigned char buf[sizeof(int)];
  int i;

  MD5_Init(&md5);
  for (i = 0; i < sizeof(int); i++) {
    buf[i] = random1 & 0xff;
    random1 = random1 >> 8;
  }
  MD5_Update(&md5, buf, sizeof(buf));
  for (i = 0; i < sizeof(int); i++) {
    buf[i] = random2 & 0xff;
    random2 = random2 >> 8;
  }
  MD5_Update(&md5, buf, sizeof(buf));
  MD5_Final(digest, &md5);
}
```

Why a Good Mixing Function is Better

Digesting 123 (7b) and 65373 (`ff5d`), extended to 128 bits

- by xor: `26ff0000000000000000000000000000`
- with MD5: `b5d8e7d2861101a0ae3dfb8520a94e7b`

Digesting 124 (7c) and 65374 (`ff5e`), extended to 128 bits

- by xor: `22ff0000000000000000000000000000` (difference is 1 bit, or 0.78%)
- with MD5: `61708b24b3b2d62d5c136779a234adc2` (difference is 71 bits, or 55.47%)

Algorithmic Attacks: LCPRNGs

This acronym stands for Linear Congruential Pseudo-Random Number Generators.

Choose integers $0 < m$, $2 \le a < m$, and $0 \le c < m$. Choose $0 \le X_0 < m$ arbitrarily. Then the "random" sequence is $X_{n+1} = (aX_n + c) \bmod m$.

Example: $m = 31$, $a = 9$, $c = 12$, $X_0 = 1$.

n	aX_n	$aX_n + c$	X_{n+1}
1	9	21	21
2	189	201	15
3	135	147	23
4	207	219	2

This sequence is periodic and the period is at most m (why).

More on LCPRNGs

Obviously, we want the period to be maximal. This can trivially be achieved with $a = 1$ and $c = 1$, but the resulting sequence is not very random-looking (why?) (That's the reason for stipulating $a \ge 2$ on the previous slide)

There is a large body of well-researched theory about LCPRNGs, how to choose a, c, and m so that random-lloking sequences emerge that pass a number of statistical tests.

Unfortunately, these numbers can still not be used for cryptographic applications, because, first of all, some parts of the numbers in LCPRNGs are periodic with small periods:

More on LCPRNGs

```
unsigned int my_random() {
  static unsigned int seed = 13579;

  return seed = 69069*seed + 314159269; /* Assume 32-bit arithmetic */
}
```

This generator gives: 13579, 1252047220, 3092059785, 2837623130, 4158474167, 323447088, 2356173845, 2574614134, 1806824227, 1286941356, 3718486113, 1977131858, 349283151, ... In hex:

k	0	1	2	3
$X_{0\cdot 4+k}$	0000350b	4aa0b974	b84d1689	a922b15a
$X_{1\cdot 4+k}$	f7dd47b7	13476930	8c705c15	99757e76
$X_{2\cdot 4+k}$	6bb1f323	4cb52aac	dda39861	75d8a352
$X_{3\cdot 4+k}$	14d1a34f	073379e8	ee0b176d	5938fbee

Subsequences of LCPRNGs

The sequence alternates between even and odd integers. Not very random...

The sequence of residues modulo 4 (the least significant two bits) is 3, 0, 1, 2, 3, 0, 1, 2..., suggesting a period of 4.

The sequence of residues modulo 8 (the least significant three bits) is 3, 4, 1, 2, 7, 0, 5, 6, 3, 4, 1, 2, 7, 0, 5, 6

In general, the least significant k bits of this generator form a periodic subsequence of length 2^k.

While the high-order bits of the generator have no such weakness, you *must not* use a LCPRNG for cryptographic applications!

Demolishing LCPRNGs

Statistical properties of the sequence (which may be OK) not so interesting. Rather, concerned with predictability of the next number in the sequence, even without knowing its internal state (which may be lousy).

A RNG is *cryptographically broken* if it is possible to predict the next n-bit number in the sequence with probability $1/2^n + \epsilon$ for some $\epsilon > 0$.

I'll show you a method to do much more: how to recover the parameters m, a, and c from the first few numbers of the sequence.

Example: The numbers 1, 68, 31, 82, 157, 192, 123, 142, 185, 188, and 87 were generated by a LCPRNG. Find m, a, and c.

Cracking LCPRNGs (1): Finding c

Assume we have numbers X_0, X_1, ... that we know are successive numbers from a LCPRNG with (so far) unknown parameters m, a, and c. How can we compute them from the numbers we have?

Assume first that a little bird has told us m. How can we compute a and c?

Assume that another little bird has also told us a. How can we compute c?

Since $X_1 \equiv aX_0 + c \pmod{m}$, we have $c \equiv X_1 - aX_0 \pmod{m}$, so knowing a and m gives us c.

Cracking LCPRNGs (2): Finding a

Next, since $X_2 \equiv aX_1 + c \pmod{m}$ and $X_1 \equiv aX_0 + c \pmod{m}$, we have $X_2 - X_1 \equiv a(X_1 - X_0) \pmod{m}$. How can we find a?

Assume $\gcd(X_1 - X_0, m) = 1$.

Answer: We use the *Extended Euclidian Algorithm*, which computes, for two positive integers M and N not only $D = \gcd(M, N)$, but also two integers A and B with $AM + BN = D$.

We find x and y so that $x(X_1 - X_0) + ym = 1$

Then, $x(X_2 - X_1)(X_1 - X_0) \equiv (X_2 - X_1)(1 - ym) \equiv (X_2 - X_1) \pmod{m}$.

Therefore, the a that we seek is $x(X_2 - X_1)$.

Excursion: Euclid's Algorithm

```
/* Given two positive integers m and n, return gcd(m,n) */
int gcd(int m, int n) {
  int r = m % n;
  while (r != 0) {
    m = n;
    n = r;
    r = m % n;
  }
  return n;
10 }
```

Euclid's Algorithm: Example

Compute $\gcd(119, 84)$ (table shows state at beginning of the while loop):

#	r	m	n
1	35	119	84
2	14	84	35
3	7	35	14
4	0	14	7

It turns out that Euclid's algorithm is *very efficient*: If n is the larger of the two algorithms, line 3 will be executed only $O(\log n)$ times.

Excursion: Extended Euclidian Algorithm

```
int egcd(int* a, int* b, int m, int n) {
  int a_prime = 1, b_prime = 0;
  int c = m, d = n;
  int q = c / d, r = c % d;

  *b = 1; *a = 0;

  /* am + bn = d; a_prime m + b_prime n = c = qd + r */
  while (r != 0) {
    int t;

    c = d; d = r;
    t = a_prime; a_prime = *a; *a = t - q*(*a);
```

```
    t = b_prime; b_prime = *b; *b = t − q*(*b);
    q = c / d; r = c % d;
    /* am + bn = d; a_prime m + b_prime n = c = qd + r */
  }

  /* am + bn = d = gcd(m, n) */
  return d;
}
```

Example: Extended Euclidian Algorithm

Here are the values at the beginning of the while loop for $m = 1769$ and $n = 551$:

#	a'	a	b'	b	c	d	q	r
1	1	0	0	1	1769	551	3	116
2	0	1	1	−3	551	116	4	87
3	1	−4	−3	13	116	87	1	29
4	−4	5	13	−16	87	29	3	0

And sure enough, $5 \cdot 1769 - 16 \cdot 551 = 29 = \gcd(1769, 551)$.

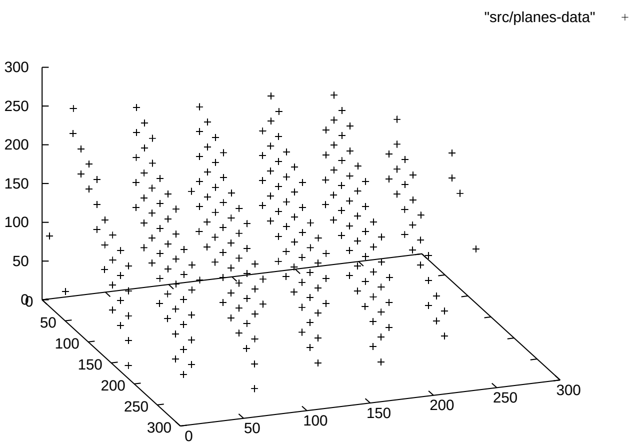
The Rain in Spain...

Donald E. Knuth was the first to notice the following phenomenon, George Marsaglia exploited it and broke LCPRNGs.

We look at a plot of (X_n, X_{n+1}) versus X_{n+2}, i.e., we take three consecutive numbers from a sequence and plot this in 3D.

Linear congruential random numbers fall mainly in the planes — George Marsaglia

...Falls Mainly in the Plains



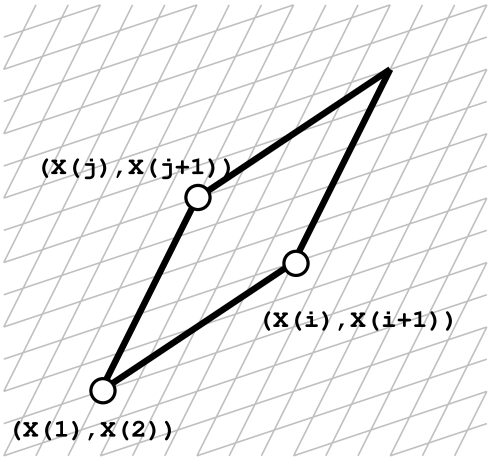
This is a plot of the sequence $X_{n+1} = (137X_n + 187) \bmod 256$. The numbers in the sequence lie in a small number of planes.

Cracking LCPRNGs (3): Finding m

It turns out that successive points of the generator lie on a *lattice*:

Cracking LCPRNGs (4): The Parallelepid

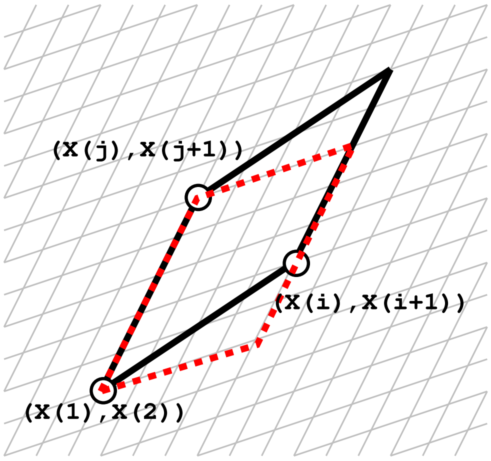
Three points made of successive values of the sequence define a parallelepid:



For a LCPRNG with modulus m, the area of the little gray cells is m (we won't prove this, and you won't be expected to prove this).

Cracking LCPRNGs (5): Shearing

The red figure's area is a multiple of the unit cell area



The transformation that transforms the black figure into the red figure (called *shearing*) is *area-preserving*.

Therefore, the area of the black figure is a multiple of m.

Cracking LCPRNGs (6): Finding the Area

Therefore, the greatest common divisor of all areas of different parallelepids is a (small) multiple of m; in practice, it is m itself.

If we want to find m, we proceed along the following lines:

```
old_gcd = gcd(d(1,2), d(2,3));
for (i = 3; ; i++) {
  int area = d(i, i+1);
  int new_gcd = gcd(old_gcd, area);

  if (new_gcd and old_gcd haven't changed since a few iterations)
    break;
  else
    old_gcd = new_gcd;
}
print(old_gcd);
```

Cracking LCPRNGs (7): Finding the Area

From linar algebra, you (should) know that the area of the parallelepid is

$$d(i,j) = \text{abs} \begin{vmatrix} X_i - X_0 & X_{i+1} - X_1 \\ X_j - X_0 & X_{j+1} - X_1 \end{vmatrix}$$

Cracking LCPRNGs (8)

We have the numbers 1, 68, 31, 82, 157, 192, 123, 142, 185, 188, and 87 and wish to know the parameters m, a, and c of the LCPRNG that generated them.

First, we find m.

We compute $d(1,2) = |(X_1 - X_0)(X_3 - X_1) - (X_2 - X_0)(X_2 - X_1)| = |67 \cdot 14 - 30 \cdot (-37)| = 2048$

$d(2,3) = |(31-1)(157-68) - (82-1)(82-68)| = 1536$

$d(3,4) = |(82-1)(192-68) - (157-1)(157-68)| = 3840$

$d(4,5) = |(157-1)(123-68) - (192-1)(192-68)| = 15104$

$d(5,6) = |(192-1)(142-68) - (123-1)(123-68)| = 7424$

Cracking LCPRNGs (9)

Compute $\gcd\big(d(1,2), d(2,3)\big) = \gcd(2048, 1536) = 512$

$\gcd\big(512, d(3,4)\big) = \gcd(512, 3840) = 256$

$\gcd(256, d(4,5)) = \gcd(256, 15104) = 256$

$\gcd(256, d(5,6)) = \gcd(256, 7424) = 256$

So it looks like $m = 256$. (Note how fast the sequence converges.)

Cracking LCPRNGs (10)

Next, we find a by computing x and y with $x(X_1 - X_0) + ym = 1$. (First we verify that $\gcd(X_1 - X_0, m) = 1$, though, but since m is a power of 2, and since $X_1 - X_0 = 67$ is odd, this is easy.)

Firing up the Extended Euclid Algorithm gives $x = 107$ and $y = -28$, hence $a = x(X_2 - X_1) \equiv 107 \cdot (31 - 68) \equiv 137 \pmod{256}$.

Cracking LCPRNGs (11)

It remains to compute c. Setting $X_1 = aX_0 + c \bmod m$ gives $c = X_1 - aX_0 \equiv 68 - 137 \cdot 1 \equiv 187 \pmod{256}$.

Now, we have $m = 256$, $a = 137$ and $c = 187$. Let's check:

k	X_k	aX_k	$aX_k + c$	X_{k+1}
0	1	137	324	68
1	68	9316	9503	31
2	31	4247	4434	82
3	82	11234	11421	157
4	157	21509	21696	192

So we were able to extract all parameters from the sequence after seeing only the first seven numbers. This is *not* confidence-inspiring!

Summary

- What is a Random Number?
- Cracking Random Number Generators
- Entropy/Workload Estimation
- *rand*(3) and Netscape's Old Generator
- Seed Guessing Attacks are Independent of the Cipher
- Use Good Mixing Functions
- How to break LCPRNGs (utterly)

References

Donald E. Knuth, *Seminumerical Algorithms*, vol. 2 of *The Art of Computer Programming*, Third Edition, Addison-Wesley, 1998.

Eastlake et. al, *Randomness Recommendations for Security*, http://www.ietf.org/rfc/rfc1750.txt.

Peter Gutmann's Web Page, http://www.cs.auckland.ac.nz/˜pgut001/.